COMMUNICATING AUTOMATA : A MODEL FOR CONCURRENT COMPUTATIONS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By

SALIL DURANI

to the

COMPUTER SCIENCE

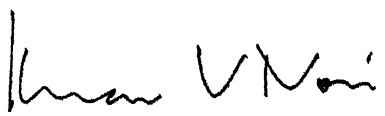
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY, 1980

C.S-1980-M-DUR-COM.

CERTIFICATE

This is to certify that the work entitled, 'COMMUNICATING AUTOMATA : A MODEL FOR CONCURRENT COMPUTATIONS', has been carried out by Sri Salil Durani under my supervision and has not been submitted elsewhere for the award of a degree.



(Kesav V. Nori)
Visiting Asst. Professor
Computer Science Programme
Indian Institute of Technology,
Kanpur

Kanpur

July, 1980

ACKNOWLEDGEMENTS

It is a great pleasure to acknowledge with gratitude, the help and guidance rendered by my thesis supervisor Prof. K.V. Nori. My sincere thanks to him for suggesting the model described in this thesis and providing sufficient inspiration to undertake and complete this work.

Thanks to the other members of the Working Group Pi viz. Prof. K.V. Nori, Mr. R. Ramaswamy, Mr. D.Ramesh Narayan, Mr. A.M.Suthar for their valuable suggestions and criticism which played an important role in the completion of this thesis. Those informal Tuesday and Friday afternoon meetings were a great source of professional satisfaction, not ignoring the valuable contributions made in ORIGAMI.

I am grateful to my colleagues Mr. R.K. Jain, Mr. Vinod Gupta and Mr. Sanjiv Kumar for providing a wonderful company in the IBM 1800 room, during the days of writing the manuscript.

Thanks are also due to Mr. Rakesh Chadha, a good friend and companion, who was a ~~ex~~constant source of inspiration and enthusiasm throughout my stay here.

Thanks to all my friends and colleagues but for whom my stay out here would not have been so pleasant and memorable.

I would like to thank Shri J. K. Misra for his excellent typing and Shri H.S.Tripathi for neat cyclostyling.

Kanpur SALIL DURANI

ABSTRACT

There are several models for concurrent processes. However, none of the ones we have seen, attempt to formally characterize the notion of concurrency. In this thesis, we attempt to construct a simple model for concurrent processes with a formal definition. One method of definition is in the form of reduction: given a collection of concurrent processes described in our model, we algorithmically reduce this collection to a single sequential process. In the reduction performed, all forms of interaction needed between concurrent processes are converted to simple actions on a global state. Several traditional and recent problems are programmed in our model to show its general applicability.

CONTENTS

CHAPTER I

INTRODUCTION

MOTIVATION:

In the short history of studies in "concurrent Computations", the landmarks (from the viewpoint of programming language features) have been:

1. Semaphores [DIJK 68]
2. Conditional Critical Regions [HOAR 72], [HANS 72]
3. Monitors [HOAR 74]
4. Path Expressions [CAMP 74]
5. Communicating Sequential Processes [HOAR 78a], [HOAR 78b]
6. Distributed Processes [HANS 78]
7. Eventcounts [REED 79].

The underlying assumptions have never been clearly stated. Attempts to design implementations for these features have led to unpleasent discoveries, hidden under the seeming simplicity purported by the feature.

One of the constant reminders regarding the dangers of absorbing concurrency as a standard feature of programming languages, is the lack of acceptance of the experimental languages even in universities. There seems to be sufficient disagreement with regard to the nature of primitives and abstraction for concurrent computations for attempting to scale the problem once again.

An initial motivation for the model of concurrency that we have developed here is the success of asynchronous hardware systems that operate with a reliability yet to be realised in Operating Systems. The loosely coupled hardware systems communicate with each other on well defined busses; they do so only when the respective subsystems are in a well defined state. An existing model that emulates this structure is Hoare's Communicating Sequential Processes [HOAR 78].

Our attempt is built around the difficulties we experienced with Hoare's model and tries to follow, more closely, the hardware designer's view. In doing so we try to be explicit and formal in our assumptions and definitions.

The model suggested is that of a Finite State Automaton (not very "far" from Regular Grammars [NORI 80]) whose transitions represent the lifetime of a concurrent process. The only peculiarity with this automaton is that it communicates, and hence the name "Communicating Automaton" (CA in short). There is a simplicity in depicting a process as a Finite State Automaton and also some of the properties applicable from Finite Automata Theory may be useful in proving properties of programs modelled in this fashion.

STRUCTURE OF THE THESIS:

In the next chapter, we formally define a Communicating Automaton in a manner analogous to Finite State Automaton. The meaning of Communication is carefully explained. Chapter III contains some traditional examples of concurrent programs (including a tough one)

to illustrate the capabilities of our model. We hope that the intuition regarding the nature of communication is clarified through these examples. In Chapter IV, we formalize our intuition about the "meaning" of communication by an algorithmic description of converting a collection of CA's to a single Finite Automaton in which no communication is needed. This construction amounts to a description of all the possibilities of simulating a collection of processes on a single processor. The thesis ends in a consolidation of the results attained and some speculation regarding future developments.

CHAPTER II

DEFINITION OF A COMMUNICATING AUTOMATON

This chapter gives a definition of the Finite State Automaton model for a concurrent process. The intention is to describe a concurrent activity as a collection of finite state automata that communicate with each other. In the examples it is assumed that a PASCAL like language is imbedded in this model.

2.1 DEFINITION:

A Communicating Automaton (CA in short) is like any other Finite State Automaton except for the fact that it has a power of communication through its specially defined Input Alphabet. It is defined as a tuple $< S, I, F, \Sigma, \delta >$ where

S = set of states of CA;

I = the initial state of CA : $I \in S$;

F = set of final states of CA : $F \subseteq S$;

Σ = input alphabet

δ = state transition function. It is a mapping which on an input symbol, transits from one state to another.

The set of states S can be partitioned into two sets S_i and S_c such that

$S = S_i \cup S_c$ and $S_i \cap S_c = \phi$ where

S_i = set of "internal" states of CA. Pictorially represented as "(S)".

S_c = set of "communicating" states of CA represented as "(S)".

The above partition is prompted from the following observation:

Consider the memory system for core memories. It is composed of a read cycle followed immediately by a write cycle. When the read cycle is over, the memory is in some well defined state that is internal to it. This state is not accessible for the purposes of any concurrent operation. After the write cycle is over, memory is once again in a communicating state.

Similarly the Input Alphabet can be partitioned into an Internal Alphabet Σ_i and a Communicating Alphabet Σ_c, such that $\Sigma = \Sigma_i \cup \Sigma_c$ and $\Sigma_i \cap \Sigma_c = \phi$.

Σ, the Input Alphabet, consists of a set of pairs of the type "P : S". The first element "P" of the pair is a guard and the second element "S" is an action. A distinction has to be made here between Σ_i and Σ_c.

Σ_i constitutes an "internal" alphabet that pertains to processing that is internal to the CA. The guards of this subset of are drawn from boolean expressions e.g. $a > 0$, $a \neq b$, $c < 2$ etc. The action parts are (a sequence of) assignment statements.

Σ_c is the "communicating" alphabet, through which a collection of CA's interact. The guards of elements of Σ_c are either input guards or output guards represented as "P?" and "Q!" respectively;

where 'P' and 'Q' are names of communication channels through which interaction takes place. The exact nature of a communication transaction is given in section 2.2 of this chapter.

The action part of elements of $\sum_c$ are: '?v' representing input transaction to result in a new value to be associated with the variable 'v' and '!E' representing an output of the value of expression 'E' respectively.

If no input or output of values is required in the action part of $\sum_c$, a pair of parentheses '( )' is used for sake of notation e.g. '!( )', '?( )'.

Also included in the action part of the Internal Alphabet, is a null command denoted as 'skip'. A null command has no effect and it never fails. Symbol 'T' denotes a boolean expression which is always 'true'.

2.2 CORRESPONDENCE:

Input and Output guards specify communication between two concurrently operating CA's. Communication occurs between the CA's whenever:

An input guard 'P?' in one CA specifies the same communication channel name 'P' as the output guard 'P!' in another CA.

In parallel with communication, input and output of values can also take place by specifying an input or an output command in the action part of the Communicating Alphabet.

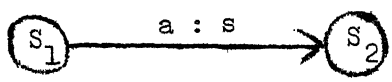
On input of a certain condition which is true of variables in the state, the CA transits to its next state, performing an assignment statement as a primitive action. Hence δ is defined as

$$\delta : S_c \times \Sigma_c \rightarrow S$$
$$: S_i \times \Sigma_i \rightarrow S$$

examples:

$$\delta(S_1, a > 0 : a := a-1) = S_2;$$
$$\delta(S_i, P?:!v) = S_j;$$

Pictorially a transition will be represented as:

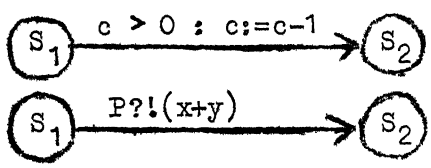


where $S_1, S_2 \in S$, $a:s \in \Sigma$

a = (input guard) |(output guard)| (boolean condition)

s = (input command) | (output command)| (primitive action)

E
examples:



There could be more than one transition emanating from a given state. If the status of variables and guards of the CA is such that a set of transitions are feasible at a particular instant of time from a particular state, one of the transitions is non-deterministically selected and others have no effect; but the choice between them is arbitrary. In an efficient implementation, an output

guard which has been ready for a long time should be favoured; but the definition of a model cannot specify this, since the relative speed of execution of the CA's is undefined.

There is an inherent non-determinism built into the model but this is in no way like non-determinism of a traditional finite state automaton. In fact this kind of non-determinism is more like Dijkstra's [DIJK 75]. In short fair play is assumed. But it is still the programmer's responsibility to prove that his program terminates correctly, without relying on the assumption of fairness in implementation.

Two transitions,"P?:!E' and 'P!:?v' are said to correspond whenever:

1. The input guard 'P?', in the transition of a CA, specifies the same signal name 'P' as in the output guard 'P!' of a transition in another concurrent CA.
2. Action symbol of one of the transitions is an input command and that of the other is an output command.
3. The action symbols corresponding to the input and output commands have compatible type definitions.

Obviously 'correspondence' has a meaning only if two CA's are in communicating states.

example:

two transitions 'P?:!E' and 'P!?v' correspond if and only if the types of expression 'E' and variable 'v', are compatible.

Transitions that correspond are executed simultaneously and the combined effect is to copy the value of expression in the output command, into the variable in the input command and simultaneously transit to the respective next states. There is no automatic buffering. In general an input or output transition is delayed until the other CA is ready with a corresponding input or output.

The communication between two CA's is from point to point. It is like communication on a bidirectional bus. Every transition between two devices on the bus has a direction which is specified by a 'master-slave' relationship.

Once a line is set up between a master and slave, a certain transaction is to be performed. This transaction is an exchange of information between a master and a slave. The direction of this exchange is to be specified. The CA which is in a state having a transition with an output guard is the 'master' e.g. 'P!'. And the one having a transition with an input guard is the 'slave' of the transaction e.g. 'P?'.

Hence 'P?:!v' specifies an output with respect to slave whereas 'P!?v' specifies an input with respect to master.

A set of concurrent CA's start executing simultaneously. Each CA, amongst this set, starts in a special 'initial' state (internal to CA) and continues to transit from state to state, performing communication whenever required. All the CA's come to a halt at the (earliest) first instance when all are in their respective

'final' states. The relative speeds with which the CA's are executing is arbitrary. The final state in this model is marked with a '*' in it.

example: '(*/S)' is a final state.

CA's that simulate processes which go on infinitely, like the Operating System, will not have any final state.

CHAPTER III

EXAMPLES

This chapter illustrates the use of a Communicating Automaton as a model for a concurrent process. Sample solutions of a variety of familiar programming exercises have been described. The simplicity of the solutions and the ease of representation is clearly depicted by the following problems and their solutions.

examples: Process Communication:

3.1 BOUNDED BUFFER:

Problem: Construct a buffering process to smooth variations in the speed of output of portions by a producer process and input by a consumer process. The buffer size 'n' is finite.

Solution: Refer to Fig. 3.1.

Discussion: There are three processes described as three CA's in Fig. 3.1. The BOUNDEDBUFFER keeps transiting from state to state, accepting a portion from a producer by means of put (put?), and giving a portion to a consumer by means of a get (get?). In either case, the exchange of a portion takes place only when the corresponding process PRODUCER and CONSUMER are ready with either a put (put!) or a get (get!) respectively. PRODUCER and CONSUMER are two cycling CA's,

the former produces and the latter consumes; and both continue to do so till an internal boolean variable 'no more' is made true through some internal means. When such a condition is true, the PRODUCER or the CONSUMER or both transit to their respective final states.

Initially the buffer is empty and so only a producer can use it. At some point in time, the buffer is full and so only a consumer can make use of it. These constraints give us an idea of three states of the BOUNDEDBUFFER viz. buffer empty 'BE', buffer full 'BF' and a third state where neither of the above is true 'NENF' (not empty not full). Thus we get, after including the internal transitions, the δ-transition mapping for the BOUNDEDBUFFER process (hence a solution), using these 3 states as the only communicating states.

All the communicating states of the BOUNDEDBUFFER are also final states, to ensure proper termination of this collection of the three CA's, whenever both PRODUCER and CONSUMER transit to their respective final states.

Also there is a possibility of a deadlock whenever either of the following conditions is true.

(i) BOUNDEDBUFFER is in the 'BE' state and there are no producers and still some consumers want to consume.

(ii) BOUNDEDBUFFER is in the 'BF' state and there are no consumers and still some producers want to produce.

3.2 SEMAPHORE:

Problem: to implement an integer semaphore, shared among an array USER (i:1..n) of client processes. Each process may increment the semaphore by means of a 'V!:!( )', or decrement it by means of a 'P!:!( )'; but the latter command must be delayed if the value of the semaphore is not positive.

Solution: Refer to Fig. 3.2.

Discussion: The CA corresponding to SEMAPHORE has two communicating states 'R' (receive) and 'SR' (send and receive). 'R' responds only to the signal 'V' and 'SR' responds to both the signals 'V' and 'P'. Both these states are also final states to ensure correct termination of this collection of CA's whenever all the client processes enter thei respective final states 'FIN'.

Remarks: This elegant and clear implementation of a general semaphore using our model, depicts that this model is (at least) as powerful as the ones cited in Chapter I; and hence sufficient to describe and solve problems in concurrent computation.

3.3 ALARM CLOCK:

Problem: An alarm clock process enables user processes to wait for different time intervals. The alarm clock receives a signal from a timer process after each time unit. The user process should be woken up after the specified time interval. This problem is due to Per Brinch Hansen [HANS 78].

Solution: Refer to Fig. 3.3.

Remarks: The problemsof representing a clock with a finite integer are ignored in this solution. All the final states are marked with a '*' to ensure correct termination criteria.

Note: The last two examples illustrate the use of this model in solving problems involving process communication and synchronization in concurrent programming.

Examples: Resource Scheduling:

3.4 READERS AND WRITERS:

Problem: Two kinds of processes, called readers and writers, share a single resource. The readers can use the resource simultaneously, but each writer must have exclusive access to it. Also further reader requests should be delayed as long as some writers are either waiting for or are using, the resource.

Solution: Refer to Fig. 3.4.

Discussion: "RWSHED" (Read write scheduler) is a CA that simulates the resource. Initially there are no read or write requests, shown as the state 'NRNWR' (no reader no writer). A reader's or a writer's request can be entertained. As long as there are no write requests, further reader requests can be entertained (depicted as 'RNWR', readers but no writers). When there are no readers, one writer is given the go ahead to use the resource, which leads to yet another state 'NRWR' (no reader but a writer). And once a writer has got hold of the

resource, no reader requests will be entertained till all the writers have finished using the resource ('W' writers only). Using these states and another communicating state where writers are entertained 'RWR', we get the complete description of 'RWSHED' as a CA. All the communicating states of this CA are also final states to ensure proper termination.

READER and WRITER are cyclic processes which keep asking for the utilization of the RESOURCE. The action 'process' indicates the processing required on what has been read and the action 'generate' indicates the necessary processing required to generate a write command. The communication channel names 'sread', 'swrite', 'fread', 'fwrite' and 'ws' stand for start read, start write, finish read, finish write and wait for signal respectively. Both the states 'NOREAD' and 'NOWRITE' are final statesof the READER and WRITER processes respectively.

3.5 ON-THE-FLY GARBAGE COLLECTION:

Problem: To develop a technique which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the process devoted to the computation proper.

Remarks: This problem has been solved by Dijkstra et al [DIJK 78] and Gries [GRIE 77] . In their solutions, the problems of termination and deadlock have been ignored. What happens if the free list is empty and we still need some free memory for computation proper? When does the garbage collector terminate? The solution given below attempts to tackle these two problems.

Abstraction of the problem as in [DIJK 78].

In an abstract form of the problem, we consider a directed graph of varying structure but with a fixed number of nodes, in which each node has at most two outgoing edges. More precisely, each node may have a left-hand outgoing edge and may have a right-hand outgoing edge, but either of them or both may be missing. In this graph a fixed set of nodes exist, called, 'the roots'. A node is called 'reachable' if it is reachable from at least one root via a directed path along the edges. The subgraph consisting of all reachable nodes and their interconnections is called 'the data structure'; non-reachable nodes i.e. nodes that do not belong to the data structure are called 'garbage nodes'. The data structure can be modified by actions of the following types:

(1) Redirecting an outgoing edge of a reachable node towards an already reachable one.

(2) Redirecting an outgoing edge of a reachable node towards a not yet reachable one without outgoing edges.

(3) Adding - where an outgoing edge was missing an edge pointing from a reachable node towards an already reachable one.

(4) Adding - where an outgoing edge was missing an edge pointing from a reachable node towards a not yet reachable one without outgoing edges.

(5) Removing an outgoing edge of a reachable node.

In actions (1), (2) and (5) nodes may be disconnected from the data

structure and thus become garbage. In actions (2) and (4) a garbage node is 'recycled' i.e. made reachable again.

Solution: Coarse Grained:

Refer to Fig. 3.5.

Conceptually the memory can be thought of as a concurrent process called 'STORE', the process of acquiring nodes for the data structure as 'MUTATOR' and the collection of garbage nodes as 'COLLECTOR'.

The MUTATOR generates the following signals:

M1 : redirect an outgoing edge of a reachable node towards an already reachable one.

M2 : 'Shade' the new target, redirect the outgoing edge of a reachable node to one that is not reachable i.e. freelist, and finally update the freelist.

E : whenever the MUTATOR process comes to an end, thus forcing termination of the COLLECTOR.

'Shading a node' means making it gray if it is white, and leaving it unchanged if it is gray or black.

The COLLECTOR implements the 'marking phase' and the collection phase. Marking will be described in terms of colors. We start with all nodes white, and will design the algorithm so that the combined activity of the collector's marking phase and the mutator will make all reachable nodes black. All nodes that are still white after the marking phase will thus be garbage.

Following predictes should be invariant:

P1 : 'during the marking phase no node will become lighter'.

P2 : 'no edge points from a black node to a white one'.

In order to make P_2 true, an additional, intermediate color 'gray' is introduced, to avoid the possibility of introducing an edge from a black node to a white one. The MUTATOR makes use of this color while introducing a new node in the data structure.

The COLLECTOR generates the following signals:

C0 : shade all roots.

C1 : Shade successors of node and blacken the node

C2 : request to return color of a node

C3 : actual communication of the color of a node.

C4 : appends a node to the free list.

C5 : makes color of a node white.

C6 : signals a deadlock to the STORE process due to a possibility of the total number of nodes available being less than what is actually required.

The termination criterion is met by defining the signals E and C6 and also introducing certain final states in the 3 CA's as shown in Fig. 3.5.

Solution : Fine Grained:

Since the problem is complex, a fine grained solution in terms of a CA is not convenient for presentation. So we adopted a

Regular Grammar notation to solve the problem. This also illustrates the fact that problems in concurrent computation are related to Regular Grammars. Certain declarations have been made for sake of completion. Non-terminals marked with a '*' are representations of final states. A number of procedures have been just named; they could be implemented in a finer detail. The meaning of these procedures is self explanatory.

Declarations:

```
Color = (w,g,b)      (* white, gray, black *)
Cell :  array [0..M] of record
                        l,r: 0..M;  (*left and right links*)
                          c: Color;
                          v: item
                        end;
root, free, nil, s, d, ss, slr : 0..M;
lr : (l,r);
flag: boolean;
```

STORE::

```
INIT ::= nil := 0; free := 2; root := 1;
         for s := 0 to 1 do
         begin cell [s] .l := 0;  cell [s ] .r := 0
               cell [s] .c := w
         end;
         for s := 2 to M do
         begin cell [ s] .l := (i+1) mod M;
               cell [s] .r := 0;
               cell [s] .c := w
         end;
         READY*.

READY*::= M1? : ? (s, lr, d); redirect (s, lr, d) : READY*
         |M2? : ? (s, lr); ALLOC.
         |C0? : ? ( ); shade-roots; READY*
         |C1? : ? s; shadesuccessorsandblacken (s); READY*
         |C2? : ? s; SEND
         |C4? : ? s; append (s); READY*
         |C5? : ? s; whiten (s); READY*
         |C6? : ? ( ); READY*
```

```
SEND ::= C3? : ! cell [ s ] .c; READY*
ALLOC::= shade (cell [s] .lr);
         redirect (s, lr, free); free := cell [free] .l ;
         cell [cell [s] .lr] .l := 0; ROUTE
ROUTE::= free = 0 : FULL  |free ≠ 0 ; READY*

FULL*::= M1? : ? (s,lr,d); redirect (s,lr,d); FULL*
        |M2? : ? (ss, slr); LOCK
        |C0? : ? ( ); shaderoots; FULL*
        |C1? : ? s; shadesuccessorsandblacken (s); FULL*
        | C2? : ? s; C3? : ! cell [s] .c; FULL*
         |C4? : ? s; append (s); READY*
        | C5? : ? s; whiten (s); FULL*
         |C6? : ? ( ); FULL*

LOCK ::= C0? : ? ( ); shaderoots; LOCK
        |C1? : ? s; shadesuccessorsandblacken (s); LOCK
        | C2? : ? s; C3! : ? cell [s] .c; LOCK
         |C4? : ? s; append (s); s := ss; lr := slr; ALLOC
        | C5? : ? s; whiten (s); LOCK
         |C6? : ? ( ); ERROR*
ERROR*
```

MUTATOR ::

```
CYCLE ::= M1 ! : ! (s,l,d); PROCESS
         |M1 ! : ! (s,r,d); PROCESS
        | M2 ! : ! (s, l); PROCESS
         |M2 ! : ! (s, r); PROCESS
        | E ! : ! ( ); FIN*
PROCESS ::= Compute; CYCLE
FIN*.
```

COLLECTOR ::

```
START ::= C0 ! : ! ( ); BEGINMARK |E? : ? ( ); FIN*
BEGINMARK ::= flag := false; MARK
MARK ::= s := 0; k := M; REPORT
REPORT ::= not flag : C6 ! : ! ( ); MARKLOOP
          |flag : flag := false; MARKLOOP
MARKLOOP ::= k > 0 : MPROCESS |k = 0 : APPEND
MPROCESS ::= E? : ?( ); FIN* | C2 ! : ! s ; MGETC
MGETC ::= C3 ! : ? c; MCHECK |E? : ? ( ); FIN*
MCHECK ::= c = gray: k := M; DOC |c ≠ gray : k := k-1; MNEXT
DOC ::= C1 ! : ! s; MNEXT |E? : ?( ); FIN*
MNEXT ::= T : s := (s+1) mod M; MARKLOOP
APPEND ::= T : s := 0; APPENDLOOP
APPENDLOOP ::= s ≤ M : APPROCESS  |s > M: MARK
APPROCESS ::= E? : ? ( ); FIN* |C2 ! : ! s ; AGETC
AGETC ::= C3 ! : ? c ; ACHECK |E? : ? ( ); FIN*
ACHECK ::= c = black : WHITEN |c = white : COLLECT
WHITEN ::= E? : ? ( ); FIN* |C5 ! : ! s ; ANEXT
COLLECT ::= E? : ? ( ); FIN* |C4 ! : ! s; flag := true; ANEXT
ANEXT ::= T : s := s+1; APPENDLOOP
```

CHAPTER IV

REDUCTION OF COMMUNICATING AUTOMATA

In this chapter an attempt has been made at formally describing the semantics of a collection of concurrent processes or CA's. Consider a system,

$$PAR = \{ CA_1 || CA_2 || \ldots || CA_n \}$$

which is a set of CA's, all executing in parallel. The aim is to clearly bring out the meaning of this parallel composition PAR.

A traditional method of defining the meaning of composite action has been to define a universal function APPLY that takes as its arguments, composite actions and their respective initial states and actually elaborates the computational process as a sequence of primitive constituent actions. This is a mechanistic definition [MCCA 60].

A recent approach, due to Backus [BACK 78] has an algebraic flavour: Two (composite) actions are considered equivalent if we can reduce one action to another through a process of algebraic simplification and substitution. Taking a cue from Backus' work, a transformation of a collection of CA's is algorithmically defined to be a single CA that has $\sum_c = \phi$ and $S_c = \phi$, i.e. it is a simple Finite State Automaton that does not communicate. The effect of this transformation is to portray the simulation of these CA's on a 'single processor'.

Each CA, CA_i, is executing on its own, modifying its state by means of actions or operations in the action part of its input alphabet. The state of the simulating construction 'Interpreter Communicating Automaton' (ICA in short) is a cartesian product of the individual states of the CA's, CA_i, $1 \leq i \leq n$.

i.e. a state 's' of ICA is defined as:

$$s = \langle s_1, s_2, \ldots, s_n \rangle$$

where, $s_i \in S_i$ {set of states of CA_i} $\forall$ $1 \leq i \leq n$

and $S_i \cap S_j = \phi$ $\forall$ $1 \leq i, j \leq n, i \neq j$

It is tacit that there is no intersection between the set of states of two CA's executing in parallel. Parallelism is possible only when there is disjointness. And anything that goes across from one machine to another is always only through a communication channel. It is values that are exchanged and not some shared storage which is being modified/manipulated. This avoids the problem of having a guardian like Monitor, Critical Region etc. for the shared storage. As long as there is no shared storage, there can be no misuse of the overall state. Avoiding the use of concepts like monitors, semaphores etc. has led to a clarity and confidence in problem solving using this model.

The meaning of the construction ICA is essentially that it comes to an 'end' whenever all CA's reach their individual final states. Effectively it is the AND condition of the finish of each of the individual CA's, which are disjoint in their state.

INTERPRETER:

Remarks: A PASCAL like notation has been used for this algorithm. All the constructs within square '[ ]' parentheses can be implemented as procedures or subroutines in some Higher Level Language. For sake of formalism, type definitions like state, vector, alphabet etc. have been introduced.

Program: INTERPRETER ( );

Notational definitions:

s(i) : a state of the i-th CA;
$s_i(j)$: initial state of j-th CA;
$x^c(i)$: a communicating state of i-th CA;
$x^i(j)$: an internal state of j-th CA;
x^c : set of $x^c(i)$;
x^i : set of $x^i(j)$;
a(i) : predicate of the input symbol of i-th CA;
: (boolean expression) | (input guard)| (output guard)
o(·) : action part of input symbol of i-th CA;
: (output command) | (input command) | (primitive action)
$\sigma(i)$: input symbol of i-th CA;
: [a(i) : o(i)] ;
T(i) : set of $\sigma(i) \subseteq \sum$;
! destination : (output command);
: action part of $\sum_c$, which is an address;
? source : (input command);
: action part of $\sum_c$, which is a value returned on evaluation of an expression;
<state> : <s(1), s(2), ..., s(n)> ;
x, x' : <state> ;
δ_i (x(i), σ(i)) : next state of i-th CA in a state x(i), on an input σ(i);
δ (x, σ) : next <state> of ICA in a <state> x, on an input σ;

```
    x_i : initial <state> of ICA;
    K   : set of  <state> ;
    D   : set of deadlock <state> ⊆ K;
    F   : set of final <state> ⊆ K;
    flag : boolean;
```

Algorithm:

```
begin  x_i := <s_i(1), s_i(2), ..., s_i(n)> ;
     [ make x_i an unmarked <state> of K ] ;
    while [there is an unmarked <state> x = <x^i U x^c> in K ] do
    begin [mark x] ;
        if ALLFINAL (x) then CONSTRUCTFINAL (x)
        else begin flag := true;
                  if x^c ≠ φ then PROCESSCOMMUNICATINGSTATES (x^c);
                  if x^i ≠ φ then PROCESSINTERNALSTATES (x^i);
                  if flag then CREATEADEADLOCKSTATE (x)
             end
    end
end;
```

Some Procedures in Detail:

```
Procedure CONSTRUCTFINAL (n);
begin
      [make x a final state in K;
       Enter x in K]
end;
procedure CREATEADEADLOCKSTATE (x);
begin
      [make x a deadlock state in K;
       Enter x in K ];
       writeln ('THERE IS A DEADLOCK')
end;
```

```
procedure PROCESSCOMMUNICATINGSTATES (x^c);
begin
    for [every pair of states x^c(i) and x^c(j) in x^c] do
    begin [let T(i) and T(j) be the sets of input symbols corresponding
           to states x^c(i) and x^c(j) respectively];
        for [every pair of input symbols σ(i) xσ(j) in T(i) x T(j)] do
        begin if MATCH (σ(i), σ(j)) then
            begin flag := false;
                [construct a new <state> x', by replacing x^c(i) and
                x^c(j) in x by δ_i (x^c(i), σ(i)) and δ_j (x^c(j), σ(j))
                respectively; let '! destination' be the action part of
                σ(i) and '? source' be the action part of σ(j);
                Define a δ-transition as δ(x, !: destination := source)
                x' = <x_1, x_2, ..., δ_i (x^c(i), σ(i))..δ_j(x^c(j), σ(j))..x_n>;
                Enter x' as an unmarked <state> in K]
            end
        end
    end
end;
procedure PROCESSINTERNALSTATES (x^i);
begin
    for [every state x(i) ε x^i] do
    begin for [every input symbol 'a' in state x(i)] do
        begin [construct a new <state> x' by replacing x(i) in
               x by δ_i (x(i), a); Define a δ-transition δ(x, a) → x';
               Enter x' as an unmarked <state> in K];
               flag := false
        end
    end
end;
```

```
function  ALLFINAL (x: <state> ): boolean;
begin
      if [all states in <state> x are final states]
          then ALLFINAL := true
          else  ALLFINAL := false
end;
function MATCH (σ(i), σ(j)); boolean;
begin
      if [the input and output transitions, on accepting input symbols
          σ (i) and σ(j), correspond ]
           then   MATCH := true
           else   MATCH := false
end;
```

Discussion:

The initial state of ICA is constructed by taking a cartesian product of the initial states of various CA's. For every internal state and for every input symbol to this state, a new <state> of ICA is constructed, defining the corresponding δ-transition. In case of a communicating state, a check is made whether there exists another communicating state of another CA, such that the input and output transitions from these states correspond. If such a match exists, a new <state> of ICA is constructed using the next states of the input and output transitions. A δ-transition is defined whose action part in the input symbol is an assignment that returns the value of the evaluated expression in the address of the destination.

A deadlock <state> is generated whenever:

(i) the states of a < state > in ICA are all communicating states and none of the transitions from these states have a match with a transition in any other CA.

(ii) $x^c \neq \phi$ and there are no transitions possible from the internal states of <state> and there exists no match between any two communicating states of x^c.

One of the advantages of this model is the fact that without actual execution, one can a priori check the existence of deadlock states and provide the simplest sequence of < state> transitions from the initial < state > that lead to a deadlock.

Whenever all the states of a <state> of ICA are final states, this <state> of ICA is added to the set of Final states of ICA.

The termination of the algorithm is guaranteed because there is an upperbound to the number of <states> of ICA and each pass through the 'while loop' marks one such <state> .

CHAPTER V

CONCLUSION

REMARKS:

The proposal made in this thesis is that input, output and communication are primitives of any concurrent programming methodology. The idea of using a Communicating Automaton as a model for concurrent computation seems to be natural; and it has given us a method for easy derivation of solutions from the specifications of the problem.

An attempt at understanding semantics of concurrent computation, has yielded as a result, the Universal Interpreter Communicating Automaton. The interpreter probably explains certain basic issues involved in simulation of parallel processes on a sequential processor e.g. multi-programming, sheduling etc.

There is a close resemblance between this model and the 'Communicating Sequential Processes' (CSP in short) of Hoare [HOAR 78a] This is in no way coincidental. Infact the model perhaps is a side effect of having examined Hoare's concepts in detail. He has introduced a language concept for concurrent processing which is suitable for a micro-computer network environment with distributed storage. The assumptions and semantics of the use of Dijkstra's guarded commands, the alternative command and the repetitive command didn't seem to be

clear and well founded. The 'termination' of a parallel command was least convincing. Not allowing output commands to appear in guards has made that model less 'structured' and more 'asymmetric'. Also execution of input and output guards in CSP is not side-effect free and hence makes the problem solution rather messy e.g. the following program segment from CSP

```
*  [P?v  →  v:=v+1
   ▯  v > 0  →  v:=v-1 ]
```

looks unpredictable in its execution. Explicit naming of processes, instead of names for communication channels in input and output commands, is not without problems either. What happens if there exists a single process which acts both as a producer and a consumer for a resource like the bounded buffer? The situation though hypothetical can't be ignored. And then it is artificial that a resource has to know the name of the process that is using it. No attempt (though possible) was made at using the model for representing subroutines, coroutines, elaborate data structure etc. as in CSP.

CSP provides a lot of flexibilities in a language which is not conceptually symmetric. Though this idea provides great verstality in the writing of programs , it may not be used very naturally. For instance in an alternative command of CSP, it is possible that one of the guards is an input guard and the other is boolean. Effectively allowing two transitions from a state, one of them leading to communication and the other one not doing so. As Hoare himself in CSP says, 'The dangers of convenient facilities are notorious'.

In yet another paper entitled, 'A Model for Communicating Sequential Processes' by Hoare [HOAR 78b], a mathematical model of the concept of a CSP is proposed. Using the notion of 'alphabet' for the 'trace' of a process, an attempt has been made at describing the language accepted by a set of parallel processes executing simultaneously. Going into issues like this has made the problem look more complex than it really is. Hence, no such attempts were made though there is ample scope for work in that direction.

CONCLUSIONS:

Work done on this model is not complete.

(i) We have not dealt with the problem of 'broadcast' communication and the kind of program structures it makes possible.

(ii) The design of another Universal Interpreter that simulates the behaviour of 'n' processes on 'm' processors (n, m ε N). Questions like the 'ordering' involved in such a multi-processor system have to be answered. The concept of 'time' is fundamental to such an issue. The relation 'happened before' has to be explained.

(iii) The issue of a detailed implementation has also been ignored. For doing that questions regarding priority schemes and scheduling involved in the so called 'non-determinism of transitions from a state' have to be answered. This would probably mean going into the design criterion of an Operating System, for instance. An abstract implementation may lead to

the examination of more appropriate architectures for concurrent processing.

(iv) A not so likeable factor may be the finite state automaton notation, which is in no way similar to normal programming languages. But such a representation allows the use of powerful results applicable in finite state automata theory. Properties like minimization of states, equivalence of two finite state automata etc. But there seems to be a problem due to the combinatorial explosion of the number of states in an Interpreter Communicating Automaton (which is unmanageable) for any large problem.

(v) In discussing this scheme for writing parallel programs, one can not forget the important issue of giving methodologies for proving properties within this scheme.

Finally, we would like to stress that this work has only proposed an abstract solution. The search is still on for a suitable real time language that could actually be seen executing our 'concurrent' problems, giving us correct solutions, without leading us into more complex problems.

REFERENCES

1. [BACK 78] Backus, John, 'Can Programming Be Liberated from the Von Neuman Style? A Functional Style and Its Algebra of Programs', 1977, Turing Award Lecture, CACM, 21, 8 (Aug. 78), pp. 613-641.

2. [CAMP 74] Campbell, R.H., and Hapermann, A.N., 'The Specificationsof Process Synchronization by Path Expressions'. Lecture Notes in Computer Science 16, Springer Verlag, 1974, pp. 89-102.

3. [DIJK 68] Dijkstra, E.W., 'Co-operating Sequential Processes', in Programming Languages (ed. F. Genuys), Academic Press, New York, 1968.

4. [DIJK 75] Dijkstra, E.W., 'Guarded Commands, Non-determinancy and Formal Derivation of Programs', CACM, 18, 8 (Aug. 75), pp. 453-457.

5. [DIJK 78] Dijkstra, E.W., et.al., 'On the Fly Garbage Collection: An Exercise in Co-operation', CACM, 21, 11 (Nov. 78), pp. 966-975.

6. [GRIE 77] Gries, David, 'An Exercise in Proving Parallel Programs Correct', CACM, 20, 12 (Dec. 1977), p. 921.

7. [HANS 72] Hansen, Per Brinch, 'Structured Multiprogramming', CACM, 15, 7 (July 1972), p. 574.

8. [HANS 78] Hansen, Per Brinch, 'Distributed Processes: A Concurrent Programming Concept', CACM, 21, 11 (Nov. 78), pp. 934-941.

9. [HOAR 72] Hoare, C.A.R., 'Towards a Theory of Parallel Programming', In Operating Systems Techniques, Academic Press, New York, 1972.

10. [HOAR 74] Hoare, C.A.R., 'Monitors: An Operating Systems. Structuring Concept', CACM, 17, 10 (Oct. 74), pp. 549-557.

11. [HOAR 78a] Hoare, C.A.R., 'Communicating Sequential Processes', CACM, 21, 8 (Aug. 1978), pp. 666-677.

12. [HOAR 78b] Hoare, C.A.R., 'A Model for Communicating Sequential Processes', Technical Report, Oxford University, December, 1978.

13. [MCCA 60] McCarthy, John, 'Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I', CACM, 4 (April 60), pp. 184-195.

14. [NORI 80] Nori, Kesav V., 'A Metalanguage for Computation with Continuation Free Semantics', Ph.D. Thesis under preparation.

15. [REED 79] Reed, D.P., and Kanodia, R.K., 'Synchronization with Eventcounts and Sequencers', CACM, 22, 2 (Feb. 79), pp. 115-123.